Rechner-Baustelle

4-Bit-Mikroprozessor selbst entwickeln, Teil 1: Planung und Konstruktion der dCPU-4

Projekt

Dietmar P. F. Möller, Christian Siemers

Was läuft eigentlich in einer CPU ab? Kann man die internen Abläufe sichtbar machen? Trotz Programmiersprachen der fünften Generation, objektorientierten Denkweisen, ausgefeilten Compiler-optimierungen: Die CPU hat sich kaum geändert. Der Artikel begleitet den (Selbst-) Bau einer 4-Bit-CPU von der Planung bis zur Hardware-umsetzung.

Prof. Dr. Dietmar P. F. Möller ist seit 1992 Ordinarius für Technische Informatik am Institut für Informatik der Technischen Universität Clausthal.

Prof. Dr. Christian Siemers ist seit 1993 Professor für Technische Informatik an der Fachhochschule Stralsund und lehrt (zeitweise) auch am Institut für Informatik der TU Clausthal.

Die Autoren forschen gemeinsam an Themen wie Hardware- und Systembeschreibungssprachen sowie Rechnerarchitekturen.

Wenn man versucht, sich die internen Vorgänge einer CPU zu verdeutlichen, was liegt da näher, als selbst eine aufzubauen? Seit der Einführung von programmierbaren Logikbausteinen lassen sich (nahezu) beliebige – wenn auch nicht beliebig viele – Logikfunktionen in einem Baustein realisieren. Also ist mit der Verwendung einiger PLDs das Ressourcenproblem zum Aufbau einer CPU gelöst.

Es gibt im wesentlichen zwei Gründe, sich mit dem Innenleben einer CPU zu befassen. Zum einen der edukative: Die Datenflüsse einer selbst aufgebauten CPU sind nun mal anschaulicher als jedes abstrakte Diagramm einer Von-Neumann-CPU. Der hier vorgestellte, diskret aufgebaute Mikroprozessor, die dCPU-4, ermöglicht letztlich sogar die Programmierung mit eigenen Ideen wie beispielsweise einem eigenen Befehlssatz.

Der zweite Grund entspringt der Praxis: Nicht jede CPU ist käuflich zu erwerben, die Vielfalt ist beschränkt. Zumindest gedanklich lassen sich Mikroprozessoren konzipieren, die beispielsweise 50 Interrupt-Quellen bearbeiten oder über spezielle problemoptimierte Befehls- und Rechensätze verfügen. Aber warum nicht die Theorie auch in die Tat umzusetzen? So wird der dCPU-4 eine integrierte Version folgen (iCPU-4). Alle Funktionen des diskreten Aufbaus sind dann in einem programmierten Logikbaustein (CPLD oder FPGA) integriert. Damit werden natürlich auch die fest verdrahteten Verbindungen zwischen den einzelnen Bausteinen leicht variierbar. Speziallösungen sind (fast) nur noch eine Frage der Ideen.

Planungsphase

Zu Beginn der konstruktiven Arbeiten steht die Planung: Was nimmt man als Ausgangspunkt, den späteren Befehlssatz, die Adressierungsmodi, die internen Register, oder gibt es allgemeine Vorgaben, die übergreifend sein können?

Aus den Erfahrungen der ersten Rechner, die im Rahmen des sogenannten Manhattan-Projekts entwickelt wurden, formulierten J. von Neumann, H. H. Goldstine und A. W. Burks 1946 anläßlich einer Bestandsaufnahme ein allgemeines Rechnermodell. Es hat bis heute – trotz aller Kritik – als Von-Neumann-Modell seine grundlegende Gültigkeit behalten (Bild 1). Danach besteht ein Rechner aus den vier Bestandteilen Control Unit (CU), Arithmetic Logical Unit (ALU) – beide zusammen bilden die Central Processing Unit (CPU) – dem Speicher und der Ein-/Ausgabeeinheit. Zusammen mit dem verbindenden Bussystem ergibt sich dabei ein optimales (im Sinne von 'minimales') System mit der Randbe-

dingung, daß Code und Daten in dem *einen* Speicher abgelegt sind [1]. Der Befehlscode wird zusätzlich als streng sequentiell betrachtet. Mit anderen Worten: sieht man von Sprungbefehlen, die die Monotonie unterbrechen, einmal ab, stehen die einzelnen kodierten Befehlsworte an monoton aufsteigenden Adressen.

Die Zentraleinheit, die CPU, der dieses Projekt gewidmet ist, beinhaltet die Steuereinheit (CU), die als Befehlsprozessorteil die komplette Hantierung aller Vorgänge um das Holen der Befehlsworte und deren Interpretation beherrschen muß, und den Datenprozessorteil (ALU), dessen Aufgabe die Berechnung einer im Befehl gewünschten Verknüpfung ist. Seit Intel 1971 mit dem 4004 den ersten 4-Bit-Mikroprozessor im Markt eingeführt hat, beinhalten die Mikroprozessoren diesen zweiteiligen Aufbau gemäß der Von-Neumann-Architektur. Er findet sich auch in diesem Projekt konsequent umgesetzt.

Architektur

Im allgemeinen kann man bei den sogenannten CISC-Architekturen (Complex Instruction Set Computer) fünf Phasen in der Befehlsverarbeitung unterscheiden, wobei selbstverständlich nicht jeder Befehl innerhalb seiner Interpretation jede Phase durchlaufen muß. Häufig werden die Programmbefehle, zur Unterscheidung im folgenden als Makrobefehle (dem in Assembler unerfahrenen Programmierer werden sie zweifellos als 'Mikro' erscheinen), durch eigene Mikroprogramme mit Programmspeicher innerhalb der CPU bearbeitet. Diese Mikroprogramme unterscheiden sich wesentgehend von Makroprogrammen, auch von in Assemblern geschriebenen, so daß es vernünftig ist, sie im Normalfall unbeeinflußbar vom Programmierer zu halten.

Die CPU-Architektur dieses Projekts läuft unter der Bezeichnung dCPU-4: Sie ist dis-kret aufgebaut und hat eine Verarbeitungsbreite von 4 Bit. Auch wenn intern kein Mikroprogramm zur Interpretation eines Makrobefehls abgearbeitet wird, ist die Ähnlichkeit mit der CISC-Struktur unübersehbar. Trotzdem bleibt die Umsetzung von Makrobefehlen in interne Abläufe gewährleistet.

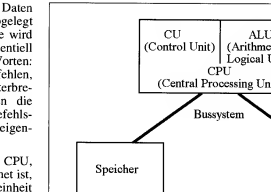

Bild 1. Das Von-Neumann-Rechnermodell.

Denn gerade letztere sollen ja transparent gestaltet werden. Zu diesem Zweck muß man sich die Phasen der Befehlsbearbeitung zu Gemüte führen:

- Fetch, Holen eines Befehls (1)
- Decode, Entschlüsseln des Befehls (2)
- Load, Laden der notwendigen Operanden (3)
- Execute, Ausführen der Operation (4) und
- Write Back, Speichern des Ergebnisses (5)

Der Befehlsprozessorteil ist für die Phasen Fetch und Decode verantwortlich, teilweise aber auch für Load und Write Back. Das Holen des Befehlscodes besteht dabei prinzipiell aus einem Speicherzugriff, für die dCPU-4 allerdings aus zweien, da jeder Befehl als 8-Bit-Code vorliegt und sich dieses Byte nur in zwei Zugriffen laden läßt.

Der geladene Befehl wird so weit dekodiert, wie es der Befehlsprozessorteil zum korrekten Laden der Operanden benötigt. Den weiteren Verlauf bestimmen an dieser Stelle die Adressierungsmodi, deren Studium für jeden Assembler-Programmierer zum Pflichtteil gehört. Dazu das folgende Beispiel:

Die *unmittelbare* Adressierung innerhalb eines Befehls bedeutet, daß der zu ladende Operand dem Befehlscode unmittelbar folgt und lediglich in einem zweiten Ladevorgang in einen geeigneten internen Speicher der CPU zu bringen ist. Eine Assembler-Kodierung wie zum Beispiel

```
LDAA # $5
```

entspricht der vorgeschlagenen Assembler-Syntax für die dCPU-4 und ist an die bekannten Sprachelemente bei der 6502/6809-CPU angelehnt. Der Befehl bedeutet demnach, daß der Befehlsprozessor zunächst den Code für 'LDAA#' (dCPU-4: $1A) in ein Befehlsregister lädt (2 Zugriffszyklen à 4 Bit), ihn interpretiert, den Befehlscodezähler – auch Programm-Counter genannt – entsprechend erhöht und schließlich auf den Operanden ($5) zugreift und diesen in den Akkumulator kopiert (1 Buszugriff à 4 Bit).

Der gleiche Befehl mit *direkter* Adressierung lautet

```
LDAA $5
```

Hier lädt der Befehlsprozessor der dCPU-4 den Code $3A für LDAA (wiederum 2 Zugriffe à 4 Bit), anschließend den Operanden $5. Besser in diesem Fall $05, denn es wird sich um eine 8-Bit-Adresse handeln (also erneut 2 Zugriffe à 4 Bit). Der geladene Operand beinhaltet jedoch nur den Zeiger, das heißt die Adresse des Speicherelements, in dem sich das eigentliche Datennibble befindet. Damit wird er also in ein Adreßzugriffsregister kopiert, um für den anschließenden Datenzugriff bereitzustehen. Im dritten Teil des Ladevorgangs wird nun via Adreßbus nicht mehr der Inhalt des Programm-Counters ausgegeben – der Codezugriff ist beendet –, sondern das Adreßzugriffsregister, in dem in diesem Beispiel $05 steht. Damit wird ein Zugriff – hier lesend – auf diese Speicherstelle angekündigt. Der Speicherinhalt bei $05 liegt anschließend am Datenbus an und wird in den Akkumulator kopiert. Erst jetzt ist der Befehl komplett abgearbeitet.

Dieses Beispiel zeigt, daß eine gewisse Teilinterpretation der Befehle bereits im Befehlsprozessor notwendig ist, um ein korrektes Laden aller Operanden zu ermöglichen. Ferner wird deutlich, daß die Adressierungsmodi (Assembler-Programmierer mögen da bitte an 'indirekt-indiziert' denken) einen großen Einfluß auf die Gestaltung dieses Prozessorteils ausüben. Die interne Systembusschnittstelle, also der Anschluß zwischen internem Adreß-, Daten- und Steuerbus einerseits und dem externen Speicher andererseits, ist für die konstruktive Unterbringung der Adressierungsmodi wichtig und wird an späterer Stelle noch im Detail behandelt.

Die richtige Adresse

An dieser Stelle sei eine kurze Einführung zu den bei dieser CPU benutzten Adressierungsmodi eingefügt: Befehle wie ROLA (Rotate Left Accumulator through Carry-Flag) enthalten bereits alle Angaben zu den benutzten Speicheroperanden, hier den Akkumulator und das Carry-Flag. Aus diesem Grund braucht in diesem einfachen Fall kein Operand geladen zu werden, man nennt diesen Adressierungsmodus 'implizit'.

Befehle, die den unmittelbar auf den Befehlscode folgenden Speicherinhalt bereits als Operand nehmen, benötigen lediglich einen weiteren (Code-) Speicherzugriff, um alle Operanden – hier natürlich nur einen – zu laden. Da der Operand unmittelbar im Code steht, bekommt diese Adressierungsart den Namen 'unmittelbar', englisch 'immediate'. Ein Beispiel dafür in der Syntax der dCPU-4 ist 'LDAA# $5', wie bereits oben dargestellt.

Enthält der Codespeicher nur die Adresse des zu ladenden oder zu speichernden Operanden, wird die Adressierungsart als 'direkt' bezeichnet. Der Inhalt der direkt benannten (Daten-)Speicherstelle ist dann als Wert zu laden. Folglich sind im Ablauf des Befehls, zum Beispiel LDAA $05, wie oben angegeben, zwei Speicherzugriffe notwendig, zunächst im Code-, dann im Datenbereich.

Für Sprungbefehle wie JMP (unbedingt) oder JNZ (bedingt, springe bzw. verzweige, wenn Zero-Flag nicht gesetzt) wird häufig eine leicht abweichende Namensgebung benutzt, wie

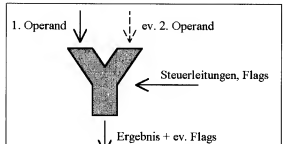

Bild 2. Prinzipieller Aufbau der Arithmetic-Logical-Unit (ALU).

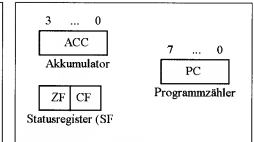

Bild 3. Das Programmiermodell.

auch in diesem Projekt. Wird der Operand (das Sprungziel) unmittelbar im Code angegeben (Jump to Location), spricht man von direkter Adressierung, obwohl dies im internen Ablauf mehr Ähnlichkeit mit der unmittelbaren Form der Datenbefehle hat. Entsprechend wird ein Sprung zu einer Zieladresse, die in einer angegebenen Adresse gespeichert wird, als 'indirekt' bezeichnet.

Die Ausführung, insbesondere bei arithmetischer oder logischer Verknüpfung, obliegt der ALU. Sind der Befehl zur weiteren Dekodierung sowie alle Operanden geladen, wird eine Verknüpfungseinheit dahingehend angesteuert, daß an ein oder zwei (gelegentlich auch mehreren) Eingangsbussen Signale anliegen, die gemäß den Vorgaben durch Steuerleitungen miteinander zu verknüpfen sind. Das Ergebnis steht dann am Ausgangsbus der ALU nach einer kurzen Zeit zur weiteren Speicherung zur Verfügung.

Die Form der Verknüpfung ergibt das typische Bild der ALU (Bild 2). Die Komplexität in der logischen Verknüpfung eines oder zweier Eingänge liegt unmittelbar in dem Befehlssatz begründet: Logisches ODER, Addition oder Bitinvertierung sind Funktionen mit einem oder zwei Parametern, die via Befehlssatz in der Programmierung eingesetzt werden und damit als Funktion in der ALU vorhanden sein müssen.

Nach diesen generellen Bemerkungen kann die eingangs gestellte Frage nach dem Ausgangspunkt der CPU-Planung beantwortet werden: Neben der allgemeinen Vorgabe, eine Von-Neumann-Architektur zu implementieren, da jede andere Architektur, siehe beispielsweise [2], zweifelsfrei den Rahmen der dCPU-4 sofort sprengen würde, gibt es noch drei weitere zu berücksichtigen:

- der Befehlssatz respektive der darin enthaltenen logischen und arithmetischen Verknüpfungen, die den Aufbau der ALU bestimmen,
- die Adressierungsmodi, die verantwortlich für wesentliche Abläufe im Befehlsprozessor sind sowie
- das Registermodell, das den Aufwand an internen Registern, Flags und Verbindungswegen bestimmt.

Normalerweise kann die Hardware zusätzlich Funktionen wie Interrupt-Request-Eingänge oder (Impuls-)Zähler bereitstellen. Das hier vorgestellte Prozessormodell muß von diesen sonst üblichen Funktionalitäten ebenso absehen wie von der Implementierung eines Stacks, da dies erheblich mehr Aufwand an Hardware innerhalb der Steuerwerks bedeutete. Als Trost sei auf die Fortführung des Projekts, die in komplexe PLDs integrierte iCPU-4, verwiesen.

In einem Punkt weicht die 4-Bit-CPU von der strengen Von-Neumann-Architektur ab: Code- und Datenspeicherbereich werden getrennt adressiert. Bereits bei der Darstellung der Adressierungsmodi ist auf den ursprünglichen Speicherbereich des Operanden hingewiesen worden. Mit einer gewissen Berechtigung läßt sich nunmehr festlegen, daß auch die Operanden der indirekten Sprünge im Datenspeicher stehen können. Daten- und Codezugriffe sind damit klar unterscheidbar. Mittels Jumper wird die dCPU-4 so konfiguriert, daß sie mit einem neunten Bit auf 'high' auf den Datenbereich, mit demselben Adreßbit auf 'low' auf den Codebereich zugreift. Dieser Kunstgriff bringt die strikte Trennung von Code- und Datenbereich und zugleich eine Verdopplung des gesamten Speicherbereichs, der mit 256 Adressen nicht gerade üppig bemessenen ist.

Verschiebebefehle	
LDAA	Load Accumulator A (direkt, unmittelbar)
STAA	Store Accumulator A (direkt)
Arithmetische Befehle	
DECA	Decrement Accumulator A (implizit)
INCA	Increment Accumulator A (implizit)
ADDA	Add with Carry to Accumulator A (direkt, unmittelbar)
CMPA	Compare with Accumulator A (direkt, unmittelbar)
Logische Befehle	
ROLA	Rotate Left Accumulator A through Carry (implizit)
EORA	Exclusive OR with Accumulator A (direkt, unmittelbar)
ORA	Logical OR with Accumulator A (direkt, unmittelbar)
ANDA	Logical AND with Accumulator A (direkt, unmittelbar)
Programmflußkontrollbefehle	
SEC	Set Carry-Flag (implizit)
CLC	Clear Carry-Flag (implizit)
JMP	Jump to Location (direkt, indirekt)
JNC	Jump if Carry not set (direkt, indirekt)
JC	Jump if Carry set (direkt, indirekt)
JNZ	Jump if Zero-Flag not set (direkt, indirekt)
JZ	Jump if Zero-Flag set (direkt, indirekt)
Sonstige Befehle	
NOP	No Operation (implizit)

Tabelle 1. Der Befehlsumfang der dCPU-4.

Das Modell

Bild 3 zeigt das Programmiermodell der dCPU-4. Es zeichnet sich durch drei Register aus: Den Akkumulator (ACC) als zentrale Speicher- und Ausgangsgröße, das Statusregister (SR) mit dem Zero-Flag (ZF) und dem Carry-Flag (CF) sowie den Programmzähler (PC). Letzter kann 8 Bit breite Werte aufnehmen, also exakt mit dem 8-Bit-Adreßbus (ohne die künstliche Erweiterung) korrespondieren.

Der Befehlssatz für die dCPU-4 ist ebenso willkürlich gewählt wie das Programmiermodell. Oberstes Prinzip für beide Konfigurationen sind Minimalität, um den Hardwareaufwand, insbesondere die Anzahl der GALs, überschaubar zu halten. Andererseits ist ein möglichst maximaler Befehlssatz notwendig, der das Schreiben von Assembler-Programmen überhaupt erst gestattet.

Tabelle 1 zeigt den kompletten Befehlssatz einschließlich der zugehörigen zugelassenen Adressierungsarten. Dem versierten Assembler-Programmierer fällt sofort auf, daß keinerlei Unterprogrammsprünge vorgesehen sind. Der Grund hierfür liegt in der (gewählten) Beschränktheit der Hardware. Unterprogrammsprünge, insbesondere die Speicherung der Rücksprungadressen, sind mit der Implementierung eines Stacks verbunden. Der jedoch fehlt in dieser CPU-Konstruktion. Trotzdem können via indirekter (unbedingter oder bedingter) Sprünge Unterprogrammkonstruktionen im Assembler geschrieben werden. Dazu jedoch erst später.

Mit dem Programmiermodell und dem Befehlssatz liegen

Mnemonic	Adressierung	Variierte Flags	Phase 1 (Fetch)	Phase 2 (Decode)	Phase 3 (Load)	Phase 4 (Execute)	Phase 5 (Write Back)	OpCode (hex)
LDAA	unmittelbar	–	*	*	* (1)	–	–	0 0 01 1010 (1Ah)
LDAA	direkt	–	*	*	* (2)	–	–	0 0 11 1010 (3Ah)
STAA	direkt	–	*	*	* (2)	*	–	0 0 11 1011 (3Bh)
DECA	implizit	Z	*	*	–	*	–	0 0 00 0000 (00h)
INCA	implizit	Z	*	*	–	*	–	0 0 00 0001 (01h)
ADDA	unmittelbar	Z, C	*	*	* (1)	*	–	0 0 01 0010 (12h)
ADDA	direkt	Z, C	*	*	* (2)	*	–	0 0 11 0010 (32h)
CMPA	unmittelbar	Z, C	*	*	* (1)	*	–	0 0 01 0011 (13h)
CMPA	direkt	Z, C	*	*	* (2)	*	–	0 0 11 0011 (33h)
ROLA	implizit	C	*	*	–	*	–	0 0 00 0100 (04h)
EORA	unmittelbar	Z	*	*	* (1)	*	–	0 0 01 0101 (15h)
EORA	direkt	Z	*	*	* (2)	*	–	0 0 11 0101 (35h)
ORA	unmittelbar	Z	*	*	* (1)	*	–	0 0 01 0110 (16h)
ORA	direkt	Z	*	*	* (2)	*	–	0 0 11 0110 (36h)
ANDA	unmittelbar	Z	*	*	* (1)	*	–	0 0 01 0111 (17h)
ANDA	direkt	Z	*	*	* (2)	*	–	0 0 11 0111 (37h)
SEC	implizit	C	*	*	–	*	–	0 0 00 1000 (08h)
CLC	implizit	C	*	*	–	*	–	0 0 00 1001 (09h)
JMP	direkt	–	*	*	* (1)	*	–	1 0 01 1100 (9Ch)
JMP	indirekt	–	*	*	* (2)	*	–	1 0 11 1100 (BCh)
JNC	direkt	–	*	*	* (1)	*	–	1 0 01 1101 (9Dh)
JNC	indirekt	–	*	*	* (2)	*	–	1 0 11 1101 (BDh)
JC	direkt	–	*	*	* (1)	*	–	1 1 01 1101 (DDh)
JC	indirekt	–	*	*	* (2)	*	–	1 1 11 1101 (FDh)
JNZ	direkt	–	*	*	* (1)	*	–	1 0 01 1110 (9Eh)
JNZ	indirekt	–	*	*	* (2)	*	–	1 0 11 1110 (BEh)
JZ	direkt	–	*	*	* (1)	*	–	1 1 01 1110 (DEh)
JZ	indirekt	–	*	*	* (2)	*	–	1 1 11 1110 (FEh)
NOP	implizit	–	*	*	–	–	–	0 0 00 1111 (0Fh)

Tabelle 2. Befehle und Adressierungen der dCPU-4. Die Zahlen in der Spalte 3 der Phase 3 geben die Anzahl der Ladezyklen an.

alle gewünschten Softwareressourcen fest. Damit kann die konstruktive Phase beginnen, die bewußt auf übergreifende Methoden wie VHDL und umfangreiche Simulationen verzichtet und statt dessen mit einer unmittelbaren konstruktiven Synthese und den zugehörigen Formulierungen der Programmierung der GALs in Assembler beginnt.

Konstruktion

Programmiermodell und Befehlssatz selbst sind dagegen – dies sei nochmals ausdrücklich erwähnt – zwar aus der Erfahrung heraus, aber dennoch willkürlich gewählt. Die Anzahl der logischen und arithmetischen Verknüpfungen beispielsweise ist mit der Programmierkapazität des GAL22V10, das die ALU bildet, limitiert. Selbstverständlich lassen sich auch komplett andere Befehlssätze implementieren. Zum Beispiel aus didaktischen Gründen, aber auch bedingt durch die Art der späteren Softwareapplikation, kann es notwendig sein, den Befehlssatz anzupassen.

Bevor die Hardwarerealisierung der Logik, die Gesamtschaltung und anschließend der interne Befehlsablauf näher im Detail beschrieben werden, erscheint eine Analyse der angegebenen Befehle und Adressierungsarten notwendig. Diese Analyse um-

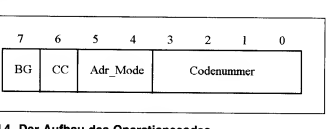

Bild 4. Der Aufbau des Operationscodes.

faßt die Mnemonics, die Phasen, die sie intern durchlaufen (von den 5 angegebenen Phasen können selbstverständlich einige entfallen), die variierten Flags sowie einen Vorschlag für die Kodierung in hexadezimaler (besser: sedezimaler) Form. Tabelle 2 zeigt dies im Überblick.

Die bezeichneten Phasen entsprechen (auch in ihrer Numerierung) den bereits oben aufgeführten. Die in der Tabelle angegebene Zahl zur Phase 3 bezieht sich auf die Anzahl der Ladevorgänge, die für das Laden des Operanden notwendig sind. Zusätzlich gilt bei der dCPU-4, daß der zweite Ladevorgang grundsätzlich im Datenspeicherbereich abläuft. Der binär und sedezimal angegebene Befehlscode wurde so gewählt, daß innerhalb von vier Bitgruppen die Kodierung sowie bereits Dekodierungsinformationen enthalten sind. Der Aufbau in dieser Form hat derart entscheidende Vorteile bei der Konstruktion der dCPU-4, daß in keiner Weise darauf verzichtet werden sollte. Die direkt als Bits kodierten Dekodierungsinformationen bedeuten zugleich eine erhebliche Aufwandsverringerung im internen Aufbau des Befehlsprozessors.

Statik

Die Zusammensetzung des Op-Codes (Bild 4) wird durch die Bitgruppen BG für Befehlsgruppe, CC für Condition Code bei bedingten Sprüngen, Adr_Mode für den Adressierungsmodus und die Befehlsnummer erreicht.

Die Befehle werden in die Gruppe der Sprungbefehle und die der andere Befehle eingeteilt. Ein Blick in Tabelle 2 zeigt, daß BG = 1 die Sprungbefehle kennzeichnet.

Bei bedingten Sprüngen wird mit CC = 1 der Sprung ausgeführt, wenn das betreffende Flag (CF oder ZF) gesetzt ist, für CC = 0 entsprechend, wenn das Flag gelöscht ist. Unbedingte Sprungbefehle sowie alle anderen Befehle setzen CC = 0.

Das Wesen der Sprungbefehle besteht darin, den Befehlszähler mit der Zieladresse neu zu laden und den nächsten Code an dieser Stelle abzulegen. Um bedingte Sprünge auszulösen, muß der Befehlsprozessor zuvor CC und das entsprechende Flag auf Gleichheit überprüfen.

Die bereits diskutierten Adressierungsarten implizit, unmittelbar (bzw. direkt) sowie direkt (bzw. indirekt für Sprungbefehle) sind in den Bits 5 und 4 kodiert, und zwar in den Bitkombinationen 00b, 01b und 11b. An dieser Stelle zeigt sich auch, daß der Modus 'direkt' bei Sprungbefehlen so kodiert wird wie 'unmittelbar' bei den übrigen und 'indirekt' entsprechend wie 'direkt'. Der eigentliche Befehl ist dann lediglich durchzumerieren – und zwar über alle Befehle durchgehend – nicht den beiden Befehlsgruppen zugeordnet. Hier ließen sich noch eine Reihe weiterer Kodierungen aufnehmen, der Phantasie des einzelnen Mikroprozessorentwicklers sind die Grenzen lediglich in der Aufnahmekapazität der GALs für die Dekodierung gesetzt. Durchnumerierung, Adressierungsart, Condition Code und Befehlsgruppe ergeben zusammengesetzt den Op-code, zum Beispiel

00111010b (3ah) für LDAA oder
10111110b (beh) für JNZ

Abschließend muß noch festgelegt werden, in welcher Reihenfolge die einzelnen Daten- und Codebytes im Speicher liegen sollen: Mit Intel als Vorbild wurde hier low Nibble vor high Nibble gewählt. Diese Reihenfolge ist natürlich variierbar, falls dieses Modell nicht den Vorstellungen entspricht. Weniger selbstverständlich ist jedoch die Tatsache, daß alle Codezugriffe in ganzzahligen Vielfachen von Bytes (und nicht Nibble) durchzuführen sind. Ein Code darf also nur bei geraden Nibble-Adressen beginnen. Im Unterschied dazu müssen Datenzugriffe natürlich auf jede Adresse möglich sein.

Im nächsten Teil dieser Artikelserie geht es um die hardwaremäßige Umsetzung der hier formulierten Voraussetzungen der dCPU-4. Die gesamte Logik, bestehend aus Interfaces, Steuerwerken, Akkumulator sowie der 'Arithmetisch-Logischen Einheit', ist in nicht weniger als zehn GALs untergebracht und wird anhand von Listings ausführlich beschrieben. *pen*

Literatur

[1] H. Bähring, 'Mikrorechnersysteme', Springer-Verlag, Heidelberg 1991

[2] A. Bode, 'Datenflußrechner: Alternative zur Von-Neumann-Architektur', ct 11/89, S. 316 ff.

Rechner-Baustelle

4-Bit-Mikroprozessor selbst entwickeln, Teil 2: Die Schaltung der dCPU-4

Projekt

Dietmar P. F. Möller, Christian Siemers

Zu Zeiten, da es noch keine programmierbaren Logikbausteine gab, wäre man nur belächelt worden, wenn man die Idee von einem selbstentwickelten Mikroprozessor geäußert hätte. Will man sich jedoch die internen Abläufe einer CPU verdeutlichen, so ist eine 'gläsern' aufgebaute Hardware allemal anschaulicher als trockene Diagramme.

Nicht nur aus edukativer Sicht macht es Sinn, sich eine CPU selbst aufzubauen. Es sind durchaus Problemstellungen denkbar, die es sinnvoll erscheinen lassen, eine CPU selbst zu entwickeln, anstatt auf ein käufliches Exemplar 'von der Stange' zurückzugreifen. Dank der Verfügbarkeit hochkomplexer PLDs beziehungsweise FPGAs stellt die Hardware heute kein Problem mehr dar. Der erste Teil des Artikels zeigte am Beispiel einer diskret aufgebauten 4-Bit-CPU die Grundlagen und Planungen, die zur Entwicklung einer eigenen CPU erforderlich sind, auf. In diesem Teil geht es nun um die schaltungstechnische Umsetzung des aufgezeigten CPU-Konzepts.

Befehlscode, Adressierungsarten und die zum kompletten Ablauf notwendigen fünf Phasen einer Befehlsausführung müssen in konsequenter Weise in dem konkreten Aufbau der CPU berücksichtigt werden. Dies geschieht in internen Daten- und Adreßpfaden sowie in zusätzlich zum Programmiermodell integrierten Registern, die dem Assemblerprogrammierer natürlich weiterhin verborgen bleiben. Bild 5 zeigt dazu das Blockschaltbild der kompletten CPU. Dieses Diagramm findet eine unmittelbare Entsprechung im Schaltplan (Bild 6), indem den Registern dort bestimmte ICs zugeordnet sind (Tabelle 3).

Innerhalb der Schaltung und damit der CPU werden mehrere interne Bussysteme benötigt, um die Aufgaben zur Abarbeitung der Makrobefehle erfüllen zu können (Tabelle 4). Neben diesen Bussystemen wird die Kommunikation und Koordination zwischen den einzelnen Elementen des Befehlsprozessors der CPU durch Steuersignale des Steuerwerks hergestellt, deren zusammenhängende Darstellung mit Bezug zu den Phasen der Befehlsabarbeitung für das Verständnis der internen Abläufe sehr wichtig ist.

Innerhalb des Steuerwerks 1 (Control Unit #1) werden mindestens 12 Phasen, teilweise sogar alle 16 Phasen benötigt, um alle Abläufe durchlaufen zu können. Nicht benötigte Phasen wie das Laden von Operanden bei Befehlen mit impliziter Adressierung bleiben in der dCPU-4 ungenutzt, da bei der Planung ein möglichst einfacher Aufbau ohne Befehlszyklusoptimierung im Vordergrund stand. Bild 7 zeigt den zeitlichen Verlauf für einen Ladebefehl mit direkter Adressierung, wobei alle mit (*) bezeichneten Phasen für andere Befehle oder Adressierungsarten gegebenenfalls einen abweichenden Verlauf nehmen. Ferner zeigt sich in diesem Timing-Diagramm, daß die Phasen sich teilweise überlappen, was den internen

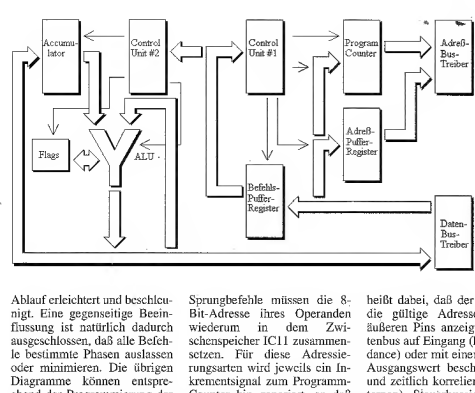

Bild 5. Das Blockschaltbild der dCPU-4.

Ablauf erleichtert und beschleunigt. Eine gegenseitige Beeinflussung ist natürlich dadurch ausgeschlossen, daß alle Befehle bestimmte Phasen auslassen oder minimieren. Die übrigen Diagramme können entsprechend der Programmierung der GALs, insbesondere des Steuerwerks 1, erstellt werden.

IC8 erzeugt die Steuersignale durch die Zählerbits aus IC7 (4 Bits, an negativer Flanke umschaltend) und den jeweils vorliegenden Befehlscodeinformationen, indem an der positiven Flanke die Signale am Ausgang des ICs übernommen werden. Durch diese Art der Synchronisierung ergibt sich ein störungsfreier Betrieb, während ohne diesen Schritt durchaus Spikes auftreten können, wie sich während der Testphase gezeigt hat.

Die Fetch-Phase läuft in Form zweier Zugriffe auf den Codespeicher ab. Der Programm-Counter (PC) gibt dazu die entsprechende Adresse aus (Steuersignal /OE_PC). Die geladenen Daten werden im Befehlszwischenspeicher 1 (GAL 18V10, IC11) zusammengesetzt (Steuersignale /IN_11, /IN_12 und /CLK_ZS1) und im Befehlszwischenspeicher 2 gespeichert (Steuersignal /LE_ZS2). Ferner fällt das scheinbare Fehlen der Decode-Phase auf. Die Dekodierung ist kompliert in der Hardware der GALs implementiert und benötigt daher keine Taktzyklen zum Ablauf.

Der erste Teil der Load-Phase lädt den auf den Befehl folgenden Operanden. Das geschieht allerdings nur, wenn eine direkte beziehungsweise indirekte Adressierung vorliegt. Direkte Nicht-Sprung- und Sprungbefehle sowie die indirekten Sprungbefehle müssen die 8-Bit-Adresse ihres Operanden wiederum in dem Zwischenspeicher IC11 zusammensetzen. Für diese Adressierungsarten wird jeweils ein Inkrementsignal zum Programm-Counter hin generiert, so daß die Adresse für den nächsten auszuführenden Befehl korrekt bleibt. Das Durchlaufen des zweiten Teils der Load-Phase – hier gibt das Adreßregister nach vorheriger Übernahme (Steuersignal /CLK_LA) die Adreßinformation aus – wird für Nicht-Sprungbefehle mit direkter Adressierung in den Phasen 12 und 13 durchgeführt, das Ladeergebnis (4 Bit) wird direkt in den Akkumulator (IC5) oder einen Zwischenspeicher (IC1) geladen. Bei indirekten Sprungbefehlen verläuft dies als zweiteiliger Zugriff (die Sprungadresse ist 8 Bit breit) in den Phasen 8…11 unter Beteiligung von IC11.

Die übrigen zwei Takte eines Befehlszyklus werden von dem Datenprozessorteil benötigt. Zu diesem Zweck schaltet das Steuerwerk 2 diverse Treiber-ICs aktiv und versorgt die ALU mit Steuersignalen bezüglich der gewünschten logischen oder arithmetischen Verknüpfung, wobei beim Ladebefehl lediglich ein Übernahmesignal (F0) zum Akkumulator (IC5) aktiviert wird. Zunächst werden jedoch die Elemente der dCPU-4 im einzelnen vorgestellt.

Das Interface zum Adreß- und Datenbus

Während der Phasen Fetch, Load sowie Write Back greift der Mikroprozessor lesend oder schreibend auf den externen Speicherbereich zu. Zugriff heißt dabei, daß der Adreßbus die gültige Adresse an den äußeren Pins anzeigt, der Datenbus auf Eingang (high impedance) oder mit einem gültigen Ausgangswert beschaltet wird und zeitlich korreliert die (externen) Steuerbussignale wie /RD und /WR aktiv sind. Die dCPU-4 verwendet dabei ein synchrones Busprotokoll für ihren Systembus, alle Signale sind taktbezogen, ein Verlängern eines Zugriffs ist nicht möglich. Dieses Busprotokoll wird im dritten Teil des Artikels näher besprochen. Der Verzicht auf modernere Varianten wie asynchron oder semisynchron [1] liegt auch an dieser Stelle in dem minimalen Aufwand an Hardwareressourcen begründet.

Die am Adreßbus ausgegebene Adresse wiederum kann durch den Programm-Counter (IC9, GAL22V10) gebildet werden – dies entspricht der Fetch- oder dem ersten Teil der Load-Phase – oder durch das Adreßpufferregister (IC10, GAL22V10), in dem die effektive Zugriffsadresse gespeichert wird, wie etwa bei direkten Zugriffen im Datenbereich. Entsprechend diesen Vorgaben muß das Register für den Programm-Counter ein Inkrement beinhalten. Programm-Counter und Adreßpufferregister müssen extern ladbar sein, eine CPU-interne Ansteuerung schaltet die Aus-

IC-Nummer	GAL-Typ	Verwendungszweck
IC2	22V10	ALU, Berechnung von Verknüpfungen
IC5	20RA10	Akkumulator und Statusflags
IC6	26CV12	Steuereinheit, Teil 2; speziell Steuerung für Datenprozessor
IC8	26CV12	Steuereinheit, Teil 1; speziell Steuerung des Befehlsprozessorteils
IC9	22V10	Programmzähler
IC10	22V10	Adreßpufferregister, Speicherung der effektiven Adresse
IC11	18V10	Befehlspufferregister, speziell für Zwischenspeicherung und für Operanden

Tabelle 3: Die in der dCPU-4 benutzten GALs und ihr Verwendungszweck.

Bussysteme	Verwendungszweck
A[0…3]	Datenbus im Datenprozessorteil (ALU, Akku, Zwischenspeicher)
B[0…3]	Datenbus zwischen Datenprozessorteil und Befehlsprozessorteil sowie externer Datenbus
C[0…1]	Steuerbus für A[0…3] und B[0…3]; C1 entspricht dem /WR-Signal (extern)
D[0…2]	Steuerbus für ALU-Operationen
E[0…3]	Datenbus zwischen Akkumulator, ALU und Treiber-IC zu Bus A[0…3]
F[0…1]	Steuerbus zum Akkumulator (Speichern etc.)
G[0…3]	Datenbus für die unteren 4 Bits des aktuellen Befehls
H[0…2]	Taktübertragung an Datenprozessorteil
I[0…7]	Datenbus für Operanden
J[0…3]	Datenbus für die oberen 4 Bits des aktuellen Befehls
K[0…7]	Adreßbus für die aktuelle Lese-/Schreibadresse
M[0…3]	Datenbus zur ALU (2. Eingangsbus)

Tabelle 4. Die internen Bussysteme der dCPU-4.

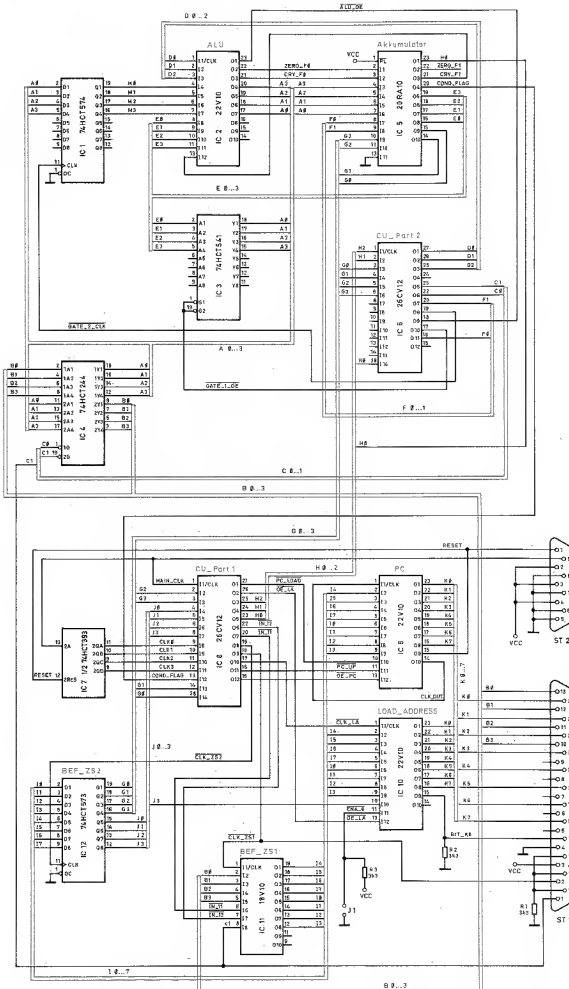

Bild 6. Schaltplan der dCPU-4.

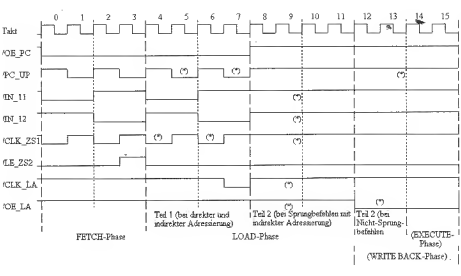

Bild 7. Der zeitliche Verlauf der Steuersignale im Steuerwerk 1.

gänge (maximal) eines dieser Register aktiv.

Listing 1 beschreibt im GAL-ASM-Format (GDS, [3]) die Programmierung des Programmzählers für die Adreßbits k0... k2 sowie die Takt- und Reset-Generierung (AR entspricht dem asynchronen Reset). Die übrigen Adreßbits sind entsprechend ausgeführt (die kompletten Listings befinden sich abrufbereit in der ELRAD-Mailbox).

Die Wahl fiel auf ein GAL-22V10, da sich hier alle Funktionen wie Aufwärtszählen, Laden eines Adreßwerts und Reset auf die Adresse 0 implementieren lassen. Andererseits ist aber auch eine eigene Taktgenerierung zur Übernahme von Zählerzuständen in die Ausgangsregister durch eine kombinatorische Logik möglich. Letzteres muß bei fertigen Zählerbausteinen entfallen.

Im Listing 1 fällt allerdings auch auf, daß der Ladevorgang einer neuen Adresse (von den Eingängen i0...i7), wie er bei Sprungbefehlen auftritt, eine Verschiebung um ein Bit mit sich führt, was einer Multiplikation mit dem Faktor zwei gleichkommt. Hier macht sich die Benutzung eines tabellengesteuerten 8-Bit-Assemblers, wie er im nächsten Abschnitt erklärt ist, bemerkbar: Der 8-Bit-Assembler berechnet die Sprungziele in 8-Bit-Adressen, die 4-Bit-CPU benötigt aber die Angabe von 4-Bit-Adressen, und diese differieren gerade um den Faktor zwei. Sollte ein anderer Assembler, vielleicht speziell für die dCPU-4, zum Einsatz kommen, dann müßte diese Linksverschiebung wieder rückgängig gemacht werden.

Der (wie erwähnt in zwei Zugriffen à 4 Bit ablaufende) Fetch des Mikroprozessors liefert einen 8-Bit-Code, der zur weiteren Dekodierung im Befehlspuffer zwischengespeichert wird. Letzteres ist durch IC11 und IC12 realisiert. Für IC11 kommt ein GAL18V10 zum Einsatz. Hierdurch ist in flexibler Weise die Zusammensetzung des Befehlscodes, ankommend in zwei Teilcodes, integrierbar, sowie als endgültigen Speicher einen 74HCT573. Listing 2 zeigt einen Ausschnitt aus der Programmierung für IC11.

Die Übernahme des Befehlscodes (während der Fetch-Phase) und eventuell des/der Operanden (während der Load-Phase(n)) vollzieht sich jeweils in zwei Abschnitten, die durch die Steuereinheit 1 geleitet werden. Die unteren 4 Bits des jeweiligen Codes werden entsprechend I0, die oberen entsprechend I4 in die Ausgangsregister geladen, wobei die Programmierung des GALs dafür Sorge trägt, daß die jeweils andere Hälfte erhalten bleibt. Nach dem kompletten Laden des Befehlscodes wird dieser in IC12 (74HCT573) übertragen. Eine Ladeadresse wird in das LOAD_ADDRESS-Register (IC10, GAL22V10), eventuelle Sprungadressen von IC11 direkt in den Programmzähler kopiert.

```
CHIP   PC   GAL22V10

CLK   i4   i5   i6   i7   i0   i1   i2   i3
/PC_LOAD PC_UP   GND
/OE   CLK_OUT   RESET   k7   k6   k5   k4   k3   k2
k1   k0   VCC

k0   :=   /k0 * /PC_LOAD
k0.TRST   =   OE

k1   :=   /k1 * k0 * /PC_LOAD + k1 * /k0 * /PC_LOAD + i0 * PC_LOAD
k1.TRST   =   OE

k2   :=   /k2 * k1 * k0 * /PC_LOAD + k2 * /k1 * /PC_LOAD
          + k2 * /k0 * /PC_LOAD + i1 * PC_LOAD
k2.TRST   =   OE

CLK_OUT   =   PC_UP

AR   =   RESET
```

Listing 1. Der Programmzähler (IC9, 22V10).

```
CHIP   BEF_ZS1        GAL18V10

CLK_ZS1   i0   i1   i2   i3   /IN_11   /IN_12   NC   NC   GND
NC   i3   i2   i1   i0   i7   i6   i5   i4   VCC

i0   :=   i0 * IN_11 + i0 * IN_12     ; Uebernahme bei IN_11,
        + i0 * /IN_11 * /IN_12        ; Halten bei IN_12 aktiv
                                      ; sowie inaktiv (bei Spikes)

i4   :=   i0 * IN_12 + i4 * IN_11     ; Uebernahme bei IN_12,
        + i4 * /IN_11 * /IN_12        ; Halten bei IN_11 aktiv
                                      ; sowie inaktiv (Spikes)
```

Listing 2. Das Befehlsregister (IC11, 18V10).

Die Adreß- und Datenbustreiber aus Bild 5 sind nicht explizit als ICs ausgeführt, sondern in den jeweiligen GALs, deren Einbeziehungsweise Ausgänge an dem jeweiligen Bus angeschlossen sind, integriert.

Das Steuerwerk 1

Die erste Phase der Code-Interpretation obliegt nun dem Steuerwerk 1 (IC8, GAL26CV12). Es entscheidet, ob und in welcher Form die Operanden zu laden sind und letztlich auch, wohin sie geladen werden müssen. Diese Decode-Phase setzt insbesondere die Bitfelder im Opcode für den Adressierungsmodus (Bit 4 und 5, siehe Bild 4, Teil 1) und die Befehlsgruppe (Sprung- oder Nichtsprungbefehl, Bit 7) in weitere interne Operationen um.

Befehle mit impliziter Adressierung benötigen keinen weiteren Operanden, die Aktivitäten am Bus werden beendet, die interne Bearbeitung beginnt. Bei unmittelbarer Adressierung von Nicht-Sprungbefehlen übernimmt, auf den Opcode folgend, entweder direkt IC1 als Zwischenspeicher oder der Akkumulator (IC5) den Inhalt der Adresse. Dies ist in jedem Fall eine Ladeoperation mit 4 Bit, begleitet von zwei PC_UP-Signalen, um den Programmzähler auf die nächste Adresse zu stellen.

Direkte Nicht-Sprungbefehle sowie alle Sprungbefehle laden die Adresse als Operand in IC11, welches sie aus zwei Ladezugriffen à 4 Bit zusammensetzt. Indirekte Sprungbefehle laden anschließend noch den Inhalt der Adresse wieder als 8-Bit-Wort in IC11. Diese Ladevorgänge steuert IC8 in Verbindung mit dem Takt, wozu die CPU insgesamt 16 Taktzyklen durchläuft. Listing 3 zeigt den relevanten Ausschnitt aus den GAL-Gleichungen.

Die Programmierung des Steuerwerks enthält neben der Generierung von Steuerleitungen zu Pufferregistern die Synthese eines Buszyklus durch insgesamt 16 Taktphasen. Dabei führt sie während jeweils zweier Phasen eine bestimmte Aktion durch (Bild 7). Beispielsweise werden in den Takten 0...3 die beiden Halbbytes des Befehlscodes in IC11 geladen, zwischen dritter und vierter Phase in IC12 gespeichert, in den Phasen 4...7 der Operand geladen, falls im Opcode Bit 4

und 5 gesetzt sowie Bit 7 gelöscht sind, oder falls Bit 7 gesetzt ist (Sprungbefehl), in den Phasen 8…11 der Inhalt der Adresse geladen, auf die der Operand verweist, falls im Opcode Bit 7 gesetzt ist. Mit Ausnahme des STAA-Befehls, der schreibend wirkt, greifen alle Nicht-Sprungbefehle mit unmittelbarer oder direkter Adressierung in der Phase 12 nochmals auf den Speicher zu: der endgültige Operand wird geladen. Das Signal CLK_ZS1 wird im übrigen zum Speicher hin als /RD-Signal herausgeführt.

Die Phasen 12…15 werden zur weiteren Durchführung der eigentlichen Operation an das Steuerwerk 2 übermittelt, mit Ausnahme der Sprungbefehle, bei denen – bei durchzuführendem Sprung, also unter Erfüllung der eventuellen Bedingung – lediglich der Programm-Counter IC9 neu zu laden ist. Diese Betrachtung der Aufgaben von IC8 macht deutlich, daß die Wahl der Kodierung mit fest zugeordneten Bits für Aktionsfelder sehr sinnvoll ist, um die Dekodierungsarbeit entsprechend zu erleichtern. Eine genaue Betrachtung der Phasen am Bus erfolgt im dritten Teil des Artikels.

```
CHIP   CD_Part1  GAL26CV12

CLK_IN   g2   g3   j0   j1   j2   VCC   j3   CLK0   CLK1
CLK2   CLK3   COND_FLAG   g1

/OB_PC  /PC_UP  /CLK_LA  /CLK_ES1 /LE_ZS2 /IN_11 GND  /IN_12   h0   h1
h2   /OB_LA   /PC_LOAD   g0


OB_PC   =   /CLK3      ; waehrend ersten 8 Phasen immer aktiv
          + CLK3 * CLK2 * /j3 * /j1 * j0
                       ; sowie bei unmittelbaren Nicht-Sprungbefehlen
                       ; in den Phasen 12 - 15

IN_11   =   /CLK1 * /CLK3   ; 0.+ 1. Phase: Laden des Befehls
              ; (Adresse: PC), sowie 4.+ 5. Phase, Laden des Operanden
              ; fuer Befehl
          + /CLK1 * /CLK2 * CLK3 * j3 * j1
              ; 8.+ 9. Phase: Laden bei direkter Adressierung eines
              ; Sprungbefehls

IN_12   =   CLK1 * /CLK3    ; 2.+ 3. Phase: Laden 2. Teil
              ; sowie 6.+7. Phase, laden 2. Teil des Operanden aus Befehl
          + CLK1 * /CLK2 * CLK3 * j3 * j1
              ; 10.+ 11. Phase bei direkter Adressierung, Sprungbefehl


CLK_ES1   =  /CLK0 * /CLK2 * /CLK3
              ; Uebernahmesignal 1. Zwischenspeicher fuer alle Befehle
          + /CLK0 * CLK2 * /CLK3 * /j3 * j1 * j0
              ; 4./5. sowie 6./7. Phase nur fuer direkte
              ; Nicht-Sprungbefehle
          + /CLK0 * CLK2 * /CLK3 * j3   ; sowie Sprungbefehle

          + /CLK0 * /CLK2 * CLK3 * j1 * j3
              ; Phasen 8/9 und 10/11 bei indirekter Adressierung von
              ; Sprungbefehlen
          + /CLK0 * /CLK1 * CLK2 * CLK3 * /j3 * j0 * /g0
          + /CLK0 * /CLK1 * CLK2 * CLK3 * j3 * j0 * /g1
              ; sowie Phase 12/13 fuer unmittelbare und direkte
              ; Adressierung bei Nicht-Sprungbefehlen, Ausnahme: STAA


PC_UP   =   CLK0 * /CLK2 * /CLK3
              ; 1. auf 2. + 3. auf 4. Phase: PC++
          + CLK0 * /CLK1 * /CLK3 * /j3 * j1 * j0
              ; Phase 5/6 und 7/8, falls direkte Adressierung bei
              ; Nicht-Sprungbefehlen vorliegt
          + CLK0 * /CLK1 * /CLK2 * CLK3 * j3 * j1 * j0
              ; dito, falls Sprungbefehl vorliegt
          + /CLK0 * CLK2 * CLK3 * j0 * /j1 * /j3
              ; Phase 12/13 und 14/15, falls unmittelbare Adressierung
              ; bei Nicht-Sprungbefehlen
          + /CLK0 * CLK1 * CLK2 * CLK3 * j3 * PC_LOAD
              ; Phase 14/15 bei Sprungbefehlen (neuen PC-Stand
              ; ueberschreiben, falls der Sprung ausgefuehrt wird)


/LE_ES2   =   CLK0 * CLK1 * /CLK2 * /CLK3
              ; 3. auf 4. Phase: Sichern Befehlswort (Latch)

CLK_LA   =   CLK0 * CLK1 * CLK2 * /CLK3 * j1 * j3
              ; 7. auf 8. Phase: Sichern Adresswort
          + CLK0 * /CLK1 * /CLK2 * CLK3 * j3 * j1 * j0
              ; 9. auf 10. Phase: Inkrement Adresse bei Jump


OB_LA   =   /CLK3 * CLK3 * j3 * j1
              ; aktiv in 8. bis 11. Phase fuer indirekte Sprungbefehle
          + CLK2 * CLK3 * /j3 * j1 * j0
              ; sowie 12.-15. Phase fuer direkte Nicht-Sprungbefehle


PC_LOAD   =  CLK2 * CLK3 * j3 * /j2 * j0 * g3 * g2
           + /g1 * /g0   ; JMP ..
           + CLK2 * CLK3 * j3 * j2 * j0 * g3 * g2          * g1 * /g0 *
COND_FLAG
              ; JZ ..
           + CLK2 * CLK3 * j3 * /j2 * j0 * g3 * g2         * g1 * /g0 *
/COND_FLAG
              ; JNZ ..
           + CLK2 * CLK3 * j3 * j2 * j0 * g3 * g2
           + /g1 * g0 * COND_FLAG
              ; JC ..
           + CLK2 * CLK3 * j3 * /j2 * j0 * g3 * g2
           + /g1 * g0 * /COND_FLAG
              ; JNC ..


h0   =   CLK2 * CLK3 * /j3              ; High fuer Takte 12-15


h1   =   /CLK0 * CLK2 * CLK3 * /j3   ; Takt 12 und 14


h2   =   /CLK1 * CLK2 * CLK3 * /j3   ; Takt 12 und 13
```

Listing 3. Das Steuerwerk 1 (IC8, 26CV12).

Der Datenprozessorteil

IC2, IC5 und IC6 bilden als ALU, Akkumulator/Statusflags und Steuerwerk 2 das Herzstück des Datenprozessors. Die übrigen ICs dienen der Zwischenspeicherung und der Abschirmung der diversen internen Datenbusse gegeneinander. IC6, das Steuerwerk 2, muß naturgemäß im Zusammenhang mit IC8 gesehen werden und erhält – wie bereits erwähnt – seine Phaseninformationen aus diesem Teil des Steuerwerks.

Das Steuerwerk 2

Der zweite Teil des Steuerwerks (IC6, GAL26CV12) erhält die notwendigen Phaseninformationen an den Pins 1, 2 und 28 (H[0…2]). Diese Informationen gehen Aufschluß darüber, ob sich die CPU in den Takten 12…15 eines Buszyklus befindet, in denen der Datenprozessorteil arbeiten soll.

Vom Opcode werden nunmehr nur noch die unteren 4 Bits (G[0…3]) benötigt, da die Adressierungsinformation bereits für die Operanden ausgewertet wurde, während das Condition-Code-Bit und die Befehlsgruppe (Bit 6 und 7 des Opcodes) nur eine Rolle bei Sprungbefehlen spielen, die jedoch im Steuerwerk 1 bearbeitet werden. Wie in Listing 3 dargestellt, werden die Phaseninformationen an das Steuerwerk 2, auch nur für Befehle aus der Nicht-Sprungbefehlgruppe übertragen.

Das Steuerwerk 2 generiert aus den Phasen- und Code-Informationen eine Reihe von Steuersignalen für die ALU (IC2). Dabei handelt es sich zum einen um solche, die der ALU die Art der Verknüpfung mitteilen, und zum anderen um welche, die innerhalb des Datenprozessors für den Datenfluß von Akkumulator und Zwischenpuffer zuständig sind. Listing 4 zeigt die komplette Verschaltung des GALs.

Die arithmetisch-logische Einheit (ALU)

Die ALU (IC2, GAL22V10) muß die Steuerinformationen in Verknüpfungen mit einem oder zwei Operanden umsetzen. Zu diesem Zweck sind vier Eingangsleitungen direkt mit dem Akkumulator (IC5) verbunden, vier weitere via einem der internen Datenbusse zum Operandenregister. Die vier Ausgangsleitungen, an denen die ALU das Ergebnis der Berechnung ausgibt, sind mit dem Akkumulator (eingangsseitig) verbunden. Dadurch wird nochmals die zentrale Funktion dieses Registers unterstrichen: das Ergebnis einer Berechnung liegt *immer* abgespeichert im Akkumulator. Weitere Ausgangsleitungen dienen der Übermittlung der Flags nach außen.

Die Konzeption der ALU ist entscheidend für die Fähigkeit der CPU, arithmetische und logische Funktionen berechnen zu können. In Anlehnung an die bekannte (aber fast verdrahtete) ALU 74xx181 stehen die Bezeichnungen d0 für M (zur Un-

d1	d2	logische Funktion (d0 = 1)	arithmetische Funktion (d0 = 0)
0	0	e XOR m	e PLUS 1 (INC e)
1	0	e ORA m	e PLUS Y PLUS C0
0	1	e AND m	e MINUS 1 (DEC e)
a1	1	ROL e	e COMPARE m

Tabelle 5. Die Funktionstabelle der ALU.

```
CHIP   CU_PART2   GAL16CV12

h2   h1   g0   g1   g2   g3   VCC   NC   NC
NC   NC   NC   NC   NC   NC
NC   /f0   /DATE_1_OE   /AUG_OR   /GATE_2_CLK
f1   GND   /c0   /c1   NC   d2   d1   d0   h0

; Die Bezeichnungen der Gates stimmen mit der Zeichnung ueberein
; Die Eingaenge gx (x = 0..7) entsprechen dem Befehlscode in
; Binaerdarstellung

f0   =     h0 * /h2 * h1 * /g3 * /g2 * /g1
                ; fuer INCA- (01h) und DECA-
                ; Befehl (00h)
         + h0 * /h2 * h1 * /g3 * g2 * /g1 * /g0
                ; fuer ADCA-Befehl (x2h) und
                ; fuer ORA-Befehl (x6h)
         + h0 * /h2 * h1 * /g3 * g2 * /g1 * /g0
                ; fuer ADLA-Befehl (04h)
         + h0 * /h2 * h1 * /g3 * g2 * g0
                ; fuer XORA-Befehl (x5h)
                ; fuer ANDA-Befehl (x7h)
         + h0 * h2 * h1 * g3 * /g2 * g1 * /g0
                ; fuer LOAA-Befehl (xah)
; f0 wird fuer den LOAA-Befehl in Phase 12 generiert,
; fuer alle anderen in Phase 14

f1   =     h0 * /h2 * h1 * /g3 * /g2 * /g1
                ; fuer INCA- (01h) und DECA-
                ; Befehl (00h)
         + h0 * h1 * /h2 * /g3 * /g2 * g1
                ; fuer ADDA (x2h) und CMPA-
                ; Befehl (x3h)
         + h0 * h1 * /h2 * /g3 * g2 * /g1 * /g0
                ; fuer ROLA-Befehl (04h)
         + h0 * /h2 * h1 * /g3 * g2 * g1 * /g0
                ; fuer ORA-Befehl (x6h)
         + h0 * /h2 * h1 * /g3 * g2 * g1 * g0
                ; fuer EORA-Befehl (x5h)
                ; fuer ANDA-Befehl (x7h)
; f1 bestimmt durch 1 die Ueberstaende der Flags aus der ALU

GATE_1_OE   =   h0 * g3 * /g2 * g1 * g0
                ; fuer STAA-Befehl (38h)
              + h0 * h2 * /h1 * /g3 * /g2 * g1 * g0
                ; und fuer CMPA-Befehl (x3h)

ALU_OE      =   h0 * /h2 * /g3 * /g2 * /g1
                ; fuer DECA-Befehl (00h) und
                ; INCA-Befehl (01h)
              + h0 * /h2 * /g3 * /g2 * g1 * /g0
                ; fuer ADDA-Befehl (x2h)
              + h0 * /h2 * /g3 * g2 * /g1 * g0      ; fuer ROLA-Befehl (04h)
                ; EORA-/ORA-Befehl (x5h/x6h)
                ; und ANDA-Befehl (x7h)

GATE_2_CLK  =  h0 * h2 * h1 * /g3 * g1
                ; fuer ADDA-Befehl (x2h),
                ; CMPA-Befehl (x3h),
                ; ORA-Befehl (x6h) und
                ; ANDA-Befehl (x7h)
              + h0 * h2 * h1 * /g3 * g2 * /g1 * g0
                ; fuer EORA-Befehl (x5h)

c0   =     h0 * h2 * /g1 * g1
                ; fuer ADDA-Befehl (x2h),
                ; CMPA-Befehl (x3h),
                ; ORA-Befehl (x6h) und
                ; ANDA-Befehl (x7h)
         + h0 * h2 * /g3 * g2 * /g1 * g0
                ; fuer EORA-Befehl (x5h)
         + h0 * h2 * g3 * /g2 * g1 * /g0
                ; fuer LDAA-Befehl (xah)

c1   =     h0 * h2 * g3 * /g2 * g1 * g0
         + h0 * /h2 * h1 * g3 * /g2 * g1 * g0
                ; fuer STAA-Befehl (3bh)
                ; in den Phasen 12 bis 14

; Es folgen die Steuerleitungen fuer die ALU

; d0 wird auf 1 fuer die logischen Funktionen der ALU gesetzt

d0   =     h0 * /g3 * g2
                ; ROLA-Befehl (04h),
                ; EORA-Befehl (x5h),
                ; ORA-Befehl (x6h) und
                ; ANDA-Befehl (x7h)

d1   =     h0 * /g3 * /g2 * g1
                ; ADDA-Befehl (x2h) und
                ; CMPA-Befehl (x3h)
         + h0 * /g3 * g2 * /g0
                ; ROLA-Befehl (04h) und
                ; ORA-Befehl (x6h)

d2   =     h0 * /g3 * /g1 * /g0
                ; DECA-Befehl (00h) und
                ; ROLA-Befehl (04h)
         + h0 * /g3 * g1 * g0
                ; CMPA-Befehl (x3h) und
                ; ANDA-Befehl (x7h)
```

```
CHL2   ADR_4_BIT_3   GAL22V10

d0   d1   d2   m0   m1   m2   m3   e0
e1   e2   e3   GND
CRY_F1   C1   C2   C3   a0   a1   a2   a3
CRY_F0   ZERO_F0        /ALU_OR        VCC

a0   =     e0 * m0 + CRY_F1 * /d0 + d1 + /d2
         + e0 * /m0 + /CRY_F1 * /d0 + d1 + /d2
         + /e0 * m0 + /CRY_F1 * /d0 + d1 + /d2
         + /e0 * /m0 * CRY_F1 * /d0 + d1 + /d2
                ; e PLUS m PLUS CRY_F1
         + /e0 * /d0 * /d1
                ; e PLUS/MINUS 1
         + e0 * /m0 * d1 * d2 * /d0
         + /e0 * m0 * d1 * d2 * /d0
                ; e und m vergleichen
         + /e0 * m0 * d0 * /d1 * /d2
         + e0 * /m0 * d0 * /d1 * /d2
                ; e XOR m
         + e0 * d0 * d1 * /d2 * m0 * d1 + /d2 * d0
                ; e OR m
         + e0 * m0 * /d1 * d2 * d0
                ; e AND m
         + CRY_F1 * d1 * d2 * d0
                ; ROL e

a0.TRST      =    ALU_OR ;

C1   =     e0 * m0 * d1 * /d2 * /d0
         + e0 * CRY_F1 * d1 * /d2 * /d0
         + m0 * CRY_F1 * d1 * /d2 * /d0
                ; C1 fuer e+m+CRY_F1
         + e0 * /d1 * /d2 * /d0
                ; C1 fuer e - 1
         + /e0 * /d1 * d2 * /d0
                ; C1 fuer e - 1

CRY_F0 =   e3 * m3 * d1 * /d2 * /d0
         * e2 * C2 * d1 * /d2 * /d0
         + m3 * C3 * d1 * /d2 * /d0
                ; CRY_F0 fuer e+m+1
         + CRY_F1 * /d1
                ; CRY_F0 fuer e+1, e-1, e AND m und e OR m
         + CRY_F1 * /d1 * /d2 * d0
                ; CRY_F0 fuer e XOR m
         + e3 * d1 * d0
                ; CRY_F0 fuer ROL
         + e3 * /m3 * d1 * d2 * /d0
         + /e3 * e2 * /m2 * d1 * d2 * /d0
         + /e3 * /e2 * e1 * /m1 * d1 * d2 * /d0
         + /e3 * /e2 * /e1 * e0 * /m0 * d1 * d2 * /d0
                ; CRY_F0 fuer Vergleich

/ZERO_F0   =   a0 + a1 + a2 + a3
```

Listing 5. Die arithmetisch-logische Einheit (IC2, 22V10).

tterscheidung zwischen arithmetischen und logischen Funktionen) sowie d1/d2 für S0/S1 zur Auswahl der jeweiligen Funktion. Tabelle 5 zeigt alle in der ALU implementierten Funktionen.

Die Wahl für den Zielbaustein fiel auf ein GAL 22V10, da dieser die größten internen Ressourcen (insgesamt 120 OR-Lines) zur Verfügung stellt, ein Kriterium, das gerade bei so vielfältigen Verknüpfungen äußerst wichtig ist. Listing 5 zeigt einen Teil der Implementierung.

Mit CRY_F1 ist dabei der Eingangswert des bisherigen Carry-Bits, mit E[0...3] und M[0...3] die Eingangsoperanden bezeichnet. Die Ausgänge C1... C3 sind lediglich als Hilfsausgänge konzipiert. Dies ist zum Beispiel für die Addition notwendig, da die Hardwareressourcen, vor allem die starre Anordnung als UND-ODER-Matrix auch des GAL22V10 ansonsten eine Implementierung solcher Funktionen unmöglich machen würde. Die Konsequenzen sind in der Realisierung der Addition durch die Serienschaltung von 1-Bit-Volladdierern sichtbar. Moderne, vor allem schnellere Konzepte wie parallele Addierer oder Carry-Look-Ahead-Addierer [4] lassen sich hier nicht berücksichtigen.

Das Ergebnis liegt nach maximal vierfacher IC-Laufzeit an den Ausgängen A[0...3], CRY_F0 für Carry-Flag und ZERO_F0 für Zero-Flag vor und kann in den Akkumulator beziehungsweise das Statusregister übernommen werden. Die Flags sind bei den unterschiedlichen Operationen nach den Angaben in Tabelle 6 definiert.

Listing 4. Das Steuerwerk 2 (IC6, 26CV12).

Der Akkumulator (IC5, GAL 20RA10) selbst ist als Register

Operation	CRY_F0
e PLUS m PLUS CRY_F1	entsprechend Ergebnis
e PLUS/MINUS 1	= CRY_F1
e XOR m, ORA, AND	= CRY_F1
ROL	= a3
e COMPARE m	= 1 fuer e >= m, sonst 0

Tabelle 6. Die Bedingungen für die Flags. Das Zero-Flag wird genau dann 1, wenn alle Ausgänge A[0...3] = 0 sind.

mit Tristate-Ausgangstreiber ohne weitere Sonderfunktionen konzipiert. Diese Funktion wäre ebenfalls zum Beispiel in einem 74HCT574 gut integriert. Da die interne Verarbeitungsbreite jedoch nur 4 Bit beträgt, konnte das Statusregister gleich mit in demselben Baustein implementiert werden.

Akkumulator und Statusregister

Die Befehle SEC und CLC, die die CPU ausführen soll, machen es jedoch notwendig, einen GAL20RA10 auszuwählen. Dieser GAL-Baustein bietet die Möglichkeit, Bedingungen zum asynchronen Reset und Preset von Ausgangsregistern zu formulieren, so daß die Makrobefehle SEC und CLC auf genau diese Funktionen zurückgeführt werden. Listing 6 zeigt wesentliche Ausschnitte der GAL-Programmierung. Dabei sollte man beachten, daß der Befehl

SEt Carry (Opcode 00001000 binär)

in einen asynchronen Reset umgesetzt wird und am Ausgang invertiert erscheint. Der Grund ist, daß das interne Register des GAL20RA10 rückgesetzt wird. Entsprechend erzeugt

CLear Carry (Opcode 00001001 binär)

einen asynchronen Preset.

Das Signal COND_FLAG wurde speziell für die Steuereinheit 1 definiert. Es übermittelt den aktuellen Wert des für einen bedingten Sprung relevanten

```
CHIP   AKKUMULATOR   GAL20RA10

/PL   ZERO_F0   CRY_F0   a3   a2   a1   a0   f0   f1
g3   g2   GND
/OE   g1   g0   e0   e1   e2   e3   COND_FLAG
CRY_F1   ZERO_F1   h0   VCC

e0      :=   a0 e0.CLCK      =   f0

e1      :=   a1
e1.CLCK      =   f0

CRY_F1      :=   CRY_F0 * f1 + CRY_F1 * /f1
                                  ; Uebernahme nur bei f1 = 1
CRY_F1.CLCK  =   f0
CRY_F1.APRST =   g3 * /g2 * /g1 * g0 * h0
CRY_F1.ARST  =   g3 * /g2 * /g1 * /g0 * h0

ZERO_F1     :=   ZERO_F0 * f1 + ZERO_F1 * /f1
ZERO_F1.CLCK =   f0
COND_FLAG    =   ZERO_F1 * g3 * g2 * g1 * /g0      ; Zero-Flag
               + CRY_F1 * g3 * g2 * /g1 * g0       ; Carry-Flag
```

Listing 6. Akkumulator/Statusregister (IC5, 20RA10).

Flags. Diese Auswertehilfe ist im Prinzip nicht Aufgabe eines Statusregisters, doch sie wurde an dieser Stelle implementiert, um den Bedarf an Eingangsverbindungen des Steuerwerks 1, niedrig zu halten.

Im dritten Teil des CPU-Projekts geht es dann um Busprotokolle, den inneren Aufbau von Makrobefehlen, die Assembler-Programmierung und schließlich um Aufbau und Inbetriebnahme der Hardware. *pen*

Literatur

[1] H. Bähring, 'Mikrosysteme', Springer-Verlag, Heidelberg 1991

[2] A. Bode, 'Alternative zur Von-Neumann-Architektur', ct 11/89, S. 316 ff.

[3] GDS Bedienungsanleitung, SH-Elektonik, Kiel 1994

[4] W. Schiffmann, R. Schmitz, 'Technische Informatik', Band 1 und 2, Springer-Verlag, Heidelberg 1992

Rechner-Baustelle

4-Bit-Mikroprozessor selbst entwickeln, Teil 3: Die internen Abläufe und die Speicherplatine

Dietmar P. F. Möller
Christian Siemers

Wer heute auf die Idee kommt, die zentrale Einheit eines Rechners 'zu Fuß' aufzubauen, kann damit nur zwei Ziele verfolgen. Entweder er will zu edukativen Zwecken die internen Abläufe einer CPU verdeutlichen, was allemal anschaulicher ist als trockene Diagramme, oder aber er möchte mit einer selbstdesignten Eigenentwicklung an Problemstellungen herantreten, die sich mit käuflichen Exemplaren nur schwerlich lösen ließen. An beide Zielgruppen wendet sich dieser Artikel.

Im ersten Teil des Rechnerprojekts ging es um die Grundlagen, die zur Planung und Entwicklung einer 'eigenen' CPU notwendig sind. Der zweite beschrieb die schaltungstechnische Umsetzung des aufgezeigten Rechnerkonzepts. Nicht weniger als sieben GALs sind notwendig, um die komplette Logik der diskret aufgebauten 4-Bit-CPU, der sogenannten dCPU-4, unterzubringen. Bild 8 zeigt die Hardware-Umsetzung in Form einer recht luftig designten Platine. In diesem Teil geht es um die Betrachtung der internen Schaltung und der internen Abläufe der CPU. Sie liefert entsprechend Aufschluß über den Anschluß externer Komponenten wie Speicher- und Peripheriebausteine. Dies ist zumeist als Businterface bekannt, und die dCPU-4 besitzt wie jede CPU drei Teilbussysteme, Adreß-, Daten- und Steuerbus, die hier an einem 25poligen SubMin-D-Stecker herausgeführt sind (Tabelle 7).

Durch einen Schaltungstrick wird in der dCPU-4 der Adreßbus als 8 + 1-Bit-Bus ausgeführt, Codezugriffe werden mit A8 = 0, Datenzugriffe mit A8 = 1 durchgeführt, was einem Adressierungsraum von 512 Adressen entspricht. Der modellhafte Charakter der CPU wird an diesem Punkt besonders fühlbar, denn ein Raum von 256 Halbbytes für die Kodierung, also 128 Programmspeicherplätze, stellt eine arge Einschränkung dar. Späteren Erweiterungen, zum Beispiel durch Integration der dCPU-4 in CPLDs oder FPGAs, soll es vorbehalten bleiben, auf einen 12-Bit-Adreßbus, der in der Interface-Definition mit A9…A11 bereits angedeutet ist, zu erweitern.

Ein aktives /RD-Signal zeigt einen Lesezugriff, ein aktives /WR-Signal entsprechend einen Schreibzugriff an. Diese Signale müssen durch eine externe Dekodierungslogik ausgewertet werden, um Speicher- und Peripherie-ICs korrekt ansteuern zu können. Der zeitliche Ablauf ist dabei streng mit dem Prozessortakt synchronisiert, also nicht mittels Wartezyklen verlängerbar, da ein semi-synchrones oder asynchrones Busprotokoll die Hardware erheblich komplexer gestalten würde (Bild 9).

Der interne Ablauf von Makrobefehlen, Teil 2

Für das Verständnis der inneren Vorgänge einer CPU ist es besonders wichtig, sich mit den internen Datenflüssen auseinanderzusetzen, da sie die Teilaktionen wie Fetch oder Load kennzeichnen. Dieser Datenfluß kann beispielsweise mit einem Oszilloskop nachgewiesen werden. Exemplarisch soll dies für den Assemblerbefehl ADDA 2ch, dessen Kodierung 32h 2ch ergibt, behandelt werden.

Bild 10 zeigt die beteiligten Einheiten an der Fetch-Phase dieses Befehls, die in den Takten 0…3 abläuft. Nach Takt 0 und 1 können an IC11 die Bitwerte 0010b an den Pins 15…12 gemessen werden, da dieses Halbbyte aus dem Codespeicher ausgelesen wurde und mittels der steigenden Flanke an /CLK_ZS1 (IC8, Pin 18) bei /IN_11 low (IC8, Pin 20) übernommen wurde.

Nach Phase 3 ist der komplette Befehlscode in IC11 mit Hilfe eines weiteren /CLK_ZS1-Impulses und des Steuersignals /IN_12 low zusammengestellt und durch /LE_ZS2 in IC12 übertragen. Der gültige Befehlscode liegt nunmehr mit Beginn der Load-Phase an IC8 zur Interpretation vor. Der Programmzähler wurde nach jedem Zugriff um eins inkrementiert und zeigt jetzt auf die Speicherstelle, an der der Operand steht.

Die Load-Phase läuft in den Takten 4…7 sowie 12 und 13 ab (Bild 11): Zunächst interpretiert die Steuereinheit 1 (IC8) die Adressierungsart als direkt

Bussystem	Bezeichnungen	Steckerbelegung
Adreßbus	A0…A8	25…17
	A9…A11	16…14 (ständig low, für späteren Ausbau)
Datenbus	D0…D3	13…10
Steuerbus	/RD	2
	/WR	1

Tabelle 7: Bussysteme der dCPU-4.

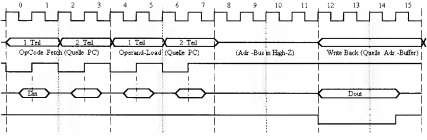

Bild 9. Zeitlicher Verlauf der Signale am Prozessorbus, hier am Beispiel des STAA-Befehls.

und stellt durch zwei Halbbyte-Zugriffe die Ladeadresse, in dem Beispiel 2ch, zusammen. Diese erste Load-Phase verläuft exakt wie die Fetch-Phase, zwei sukzessive Zugriffe werden durch /CLK_ZS1, /IN_11 und /IN_12 intern gesteuert, der PC wird insgesamt um zwei erhöht. Das Ladeergebnis wird beim Übergang Takt 7 auf 8 allerdings in IC10 kopiert, da der Datenzugriff auf diese Adresse gehen soll.

Stückliste

dCPU-4

Halbleiter:

IC1	74HCT574
IC2,9,10	GAL 22V10
IC3	74HCT541
IC4	74HCT244
IC5	GAL 20RA10
IC6,8	GAL 26CV12
IC7	74HCT393
IC11	18V10

Sonstiges:

R1...3	Widerstände 3k3
12 × CBKondensatoren100n, ker.	
J1	Jumper, 2pol.
ST1	25pol. Sub-D-Buchse, gewinkelt
ST2	9pol. Sub-D-Buchse, gewinkelt

1 Platine

Im Verlauf der Takte 8...11 dieses Befehls verweilt die dCPU-4 im inaktiven Zustand, alle Bustreiber sind auf High-Z geschaltet. In den Takten 12 und 13, der zweiten Load-Phase (Bild 12), wird der Inhalt des Adreß-Pufferregisters, IC10, am Adreßbus ausgegeben und /CLK_ZS1 im Takt 12 auf Low gesetzt: Der Lesezugriff auf die Adresse 2ch beginnt, /CLK_ZS1 hat nach außen hin die Funktion des /RD-Signals. Dieser Lesezugriff ist nur 4 Bit breit, erfolgt also nur einfach, das Ergebnis wird in ein Zwischenregister (IC1) am Eingang der ALU kopiert.

Die abschließenden Phasen 14 und 15 genügen als Execute-Phase der ALU, um das Ergebnis zu berechnen und in den Akkumulator einschließlich der Statusflags zu schreiben (Bild 13). Zu diesem Zweck sind alle Eingangsinformationen (bisheriger Inhalt des Akkumulators, gelesenes Daten-Nibble, Flags) vorhanden, die Steuerinformationen werden durch die Steuereinheit 2 (IC6) generiert, die ihre Taktinformationen durch IC8 und die Informationen zum Befehlscode durch IC12 erhält. Mit dem Übergang von Takt 14 auf 15 erhält der Akkumulator samt Flags durch das Steuersignal f0 (IC6, Pin 16) den notwendigen Übernahmetakt; die Flags, die beim Additionsbefehl natürlich mit übernommen werden, werden allerdings nur für f1 = 1 neu gesetzt und behalten ansonsten ihren Wert.

Die Speicherplatine

Zum Betrieb der dCPU-4 sind noch zwei weitere Platinen notwendig: eine Speicherplatine sowie eine Taktplatine. Die Speicherplatine enthält ein EPROM mit 32 kByte Programmspeicherplatz, ein statisches RAM mit 8 kByte, ein 8-Bit-Eingabe- und 8-Bit-Ausgabeport sowie eine Dekodier- und Umschaltelogik (Bild 14). Die Speicher-ICs sind angesichts des begrenzten Adressierungsbereichs überdimensioniert, was aber zumindestens für das EPROM durch eine Umschaltelogik mit mehreren speicherbaren Programmen gut ausgenutzt werden kann.

Das EPROM, IC1, wird mit den Adreßbits A1...A10 des (erweiterten) Adreßbus der dCPU-4 verbunden, und zwar an den Eingängen für A0...A9. Das Bit A0 selbst dient der Selektierung zwischen Low Nibble und High Nibble während eines Codezugriffs, das wiederum durch IC6 der Speicherplatine gesteuert wird. Durch diese Schaltung ist

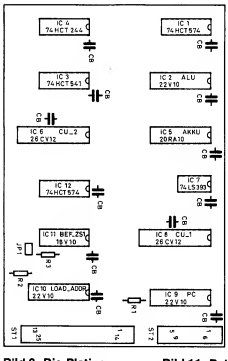

Bild 8. Die Platine der dCPU-4.

Bild 10. Daten- und Steuersignalfluß während der Fetch-Phase.

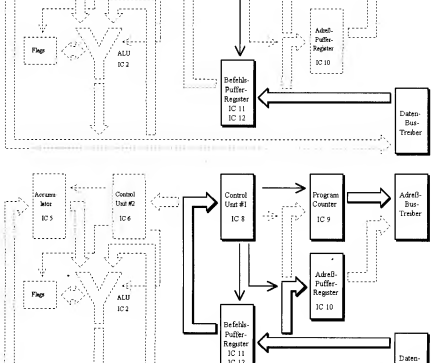

Bild 11. Daten- und Steuersignalfluß während der ersten Load-Phase.

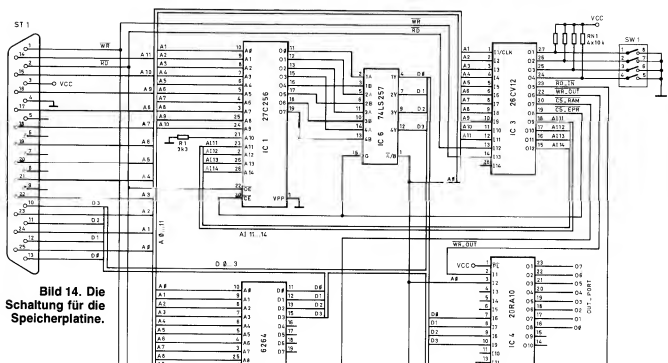

Bild 14. Die Schaltung für die Speicherplatine.

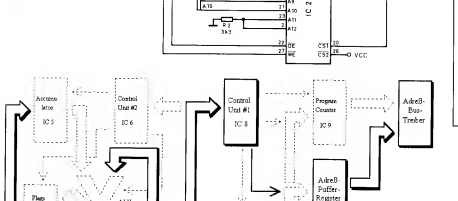

Bild 12. Daten- und Steuersignalfluß während der zweiten Load-Phase.

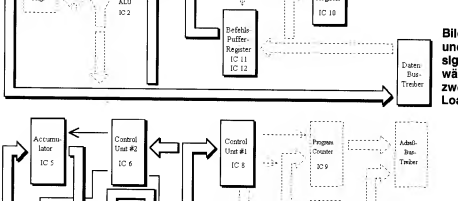

Bild 13: Daten- und Steuersignalfluß während der Execute-Phase.

die Verwendung eines 8-Bit-Assemblers, wie er im nächsten Teil vorgestellt werden wird, möglich, ohne den erzeugten Code in einzelne Halbbytes aufteilen zu müssen (der Hinweis zur Berechnung der Sprungadressen aus dem zweiten Teil dieser Artikelserie bleibt gleichwohl erhalten).

Diese Beschaltung des EPROMs ist praktikabel, da dieser Speicher nur ausgelesen wird. Im Fall des RAMs muß allerdings auch mit Schreibzugriffen gerechnet werden, und dann müßte ein Schreibzugriff auf das obere Nibble eines Datenbytes durch einen Leszugriff auf das untere Nibble mit gleichzeitigem Rückschreiben begleitet werden: Ein zwar lösbares Problem eines kleinen Zustandsautomaten, aber angesichts einer sehr viel einfacheren Lösungsmöglichkeit wurde hier darauf verzichtet: Das RAM, IC2, wird nur in den unteren 4 Bits benutzt.

Der Ein- und Ausgabeport besteht jeweils aus einem GAL20RA10 mit entsprechender Programmierung. Eine Zwischenspeicherung am Eingabeport geschieht nicht, die Daten können zwar in zwei Schritten eingelesen werden, müssen aber über beide Lesezyklen konstant bleiben. Der Ausgangs-20RA10, IC5, kann eine LED treiben, die zum Beispiel über einen 1-kΩ-Widerstand an +5 V geschaltet ist und genau dann leuchtet, wenn der Ausgang auf Low geschaltet ist.

Der nächste Teil bringt die noch fehlende Hardware (Taktplatine) und beschäftigt sich mit dem schon erwähnten 8-Bit-Assembler.

roe

Rechner-Baustelle

4-Bit-Mikroprozessor selbst entwickeln, Teil 4: Die Taktplatine und die Assembler-Programmierung

Dietmar P. F. Möller, Christian Siemers

Bisher zeigte der Artikel, wie man mit das Konzept einer CPU selbst entwickelt und in konkrete Schaltungstechnik überführt. Eine Speicher- und I/O-Platine sorgt für die notwendige Kommunikationsfähigkeit mit der Außenwelt. Um dem Kleinrechner aber auch Leben einzuhauchen, fehlen ihm noch eine Takt-Platine als Motor und natürlich Software. Als Programmierumgebung bieten sich hier tabellengesteuerte 8-Bit-Assembler an.

Die Speicherplatine zur 4-Bit-CPU (Bild 15) enthält neben einem EPROM zur Unterbringung eines oder mehrerer Programme und einem statischen RAM einen 8 Bit breiten I/O-Port sowie eine Dekodier- und Umschaltlogik (Bild 14, ELRAD 12/94). Ein- und Ausgabeport bestehen jeweils aus einem GAL20RA10 (IC4, IC5). Listing 7 und 8 zeigen die relevanten Teile der Programmierung dieser Bausteine. Die Wahl fiel auf GALs vom Typ 20RA10, da diese eine große Flexibilität insbesondere bei der Taktdefinition besitzen. Dies kann beispielsweise in erweiterter Programmierung ausgenutzt werden.

Je zwei der Output-Leitungen des IC5 sind fest miteinander verdrahtet. Die Programmierung bewirkt allerdings, daß maximal eine der Leitungen treiben kann. Die Ausgangstreiber sind im übrigen nur aktiv, wenn der Pin 13 des GAL (/RD_IN) auf Low liegt.

Die Lage der Speicher- und I/O-Bereiche wird durch das GAL IC3 bestimmt, das auch zugleich eine Auswahl aus den möglichen Programmbereichen mit Hilfe von vier Schaltern (SW1) ermöglicht. Die Wahl fiel hier auf ein GAL26CV12, da dieses zwölf Output-Leitungen besitzt. Die Programmierung läßt sich vielen Gegebenheiten anpassen. Die im Listing 9 vorgeschlagene Version bietet von 00h…0ffh 256 Halbbytes (also 128 Bytes) Programmspeicherplatz (EPROM), von 100h…1fdh 254 RAM-Speicherzellen und bei 1feh sowie 1ffh die Ein- oder Ausgabeports, Low und High Nibble, zwischen denen je nach Lese- oder Schreibrichtung umgeschaltet wird. Das Adreßbit A8 wird bei Datenspeicherzugriffen automatisch auf '1' generiert, wenn auf der dCPU-4 der Jumper JP1 gesteckt ist.

Immer im richtigen Takt

Bild 16 zeigt die Schaltung der Taktplatine. Ein Quarzgenerator erzeugt zusammen mit einem Zählerbaustein vom Typ 74HCT93 (IC2) einen teilbaren Takt. Mittels DIP-Schalter SW1 lassen sich Taktraten von 1 MHz, 500 kHz (1:2), 125 kHz (1:8) und 62,5 kHz (1:16) wählen bezichungsweise auf Handtakt umstellen. Mit Hilfe des manuellen Taktsignals lassen sich die Phasen der CPU einzeln durchlaufen und so die Signalwege genau verfolgen. Die Generierung übernehmen zwei Gatter eines 7400 (IC1) in Zusammenarbeit mit einem Taster (S2). Das Reset-Signal wird mit den beiden verbleibenden Gattern aus IC1 sowie S1 generiert. Ein 9poliges Kabel stellt die Verbindung mit der dCPU-4 her. Es kann gleich auch die Stromversorgung der CPU übernehmen. An dieser Stelle sei daraufhingewiesen, daß die Numerierung des Steckers ST2 im Schaltplan der CPU, Bild 6 in ELRAD 11/94, falsch ist. Pin 5 des Steckers – das Reset-Signal – muß in der Zeichnung oben liegen, Pin 1 unten.

Der Strombedarf der dCPU-4 einschließlich der Takt- und Speicherplatine ist, bedingt durch die eingesetzten GALs und deren Verlustleistung, recht groß. Es sollte eine Spannungsversorgung von 5 V/2 A zur Verfügung stehen, um einen einwandfreien Betrieb zu gewährleisten.

Das Einschaltverhalten der dCPU-4 entspricht dem aller CPUs. Das Reset-Signal muß so lange gehalten werden, bis der Takt einen stabilen Zustand erreicht hat und kann dann zurückgenommen werden. Nach dem Einschalten der Versorgungsspannung empfiehlt es sich also unbedingt, das Reset-Signal nochmals auszulösen. Im Handbetrieb muß der Reset-Knopf sogar gehalten werden, während per Hand ein Taktimpuls ausgelöst wird. Durch diesen Kunstgriff setzt die CPU, unabhängig vom Einschaltverhalten bei Anlegen der Betriebsspannung, garantiert alle Register auf einen gültigen Wert und startet korrekt.

Die Assemblerprogrammierung

Nach vielen Details zum internen Aufbau fehlt eigentlich nur noch die Programmierung. An dieser Stelle sei nochmals darauf verwiesen, daß die konkreten Befehle, die der Assemblerprogrammierer nutzen kann, eine Folge des internen Aufbaus einschließlich der Programmierung der GALs sind (siehe hierzu Tabelle 2 im ersten Teil des Artikels).

Als Übersetzungsprogramm bieten sich tabellengesteuerte 8-Bit-Assembler an. Teilweise sind diese sogar alsShareware-

Programme erhältlich. So beispielsweise der von Speech Technology Inc., Software Division, 837 Front Street South, Issaquah, WA 98027. Eine für die dCPU-4 gültige Übersetzungstabelle gehört zum Lieferumfang der Platine und ist auch in der ELRAD-Mailbox zu finden (TASMD4.TAB).

Die Assemblerprogramme für die dCPU-4 werden nach den allgemein üblichen Regeln geschrieben und übersetzt. Hierzu gehören Definitionen von Variablen beziehungsweise Speicherplätzen, Sprungmarken, Statthalter für aktuelle Werte zum Beispiel des Programmzählers ebenso wie bedingte

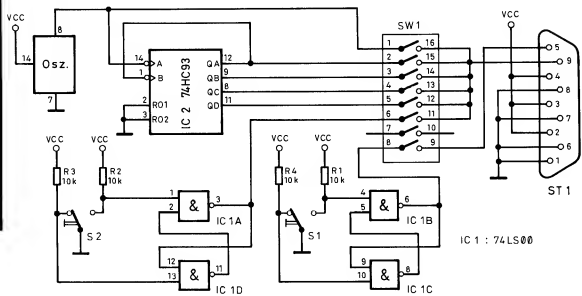

Bild 16. Auf der Taktplatine wird neben den verschiedenen Taktraten auch das Reset-Signal erzeugt.

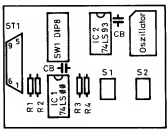

Bild 17. Der Motor der CPU: die Taktplatine.

Stückliste

Taktplatine

Halbleiter

IC1	74LS00
IC2	74LS9

Sonstiges

R1...4	10k
2 × CB	100n, ker.
Oszillator	Quarzoszillator, 1 MHz
SW1	DIP-Switch 8
S1,2	Minitaster, 2pol., Um
ST1	Sub-D-Buchse, weibl.
1 Platine	

Stückliste

Speicherplatine

Halbleiter

IC1	EPROM 27C256
IC2	RAM 6264
IC3	GAL 26CV12
IC4,5	GAL 20RA10
IC6	74LS257

Sonstiges

R1,2	Widerstände 3k3
RN1	Widerstandsnetzwerk 4 × 10k
5 × CB	Kondensatoren, 100n, ker
SW1	4fach Dil-Schalter
ST1	25pol. Sub-D-Buchse, gewinkelt
2 × Steckerleiste, 1 × 8pol.	
1 Platine	

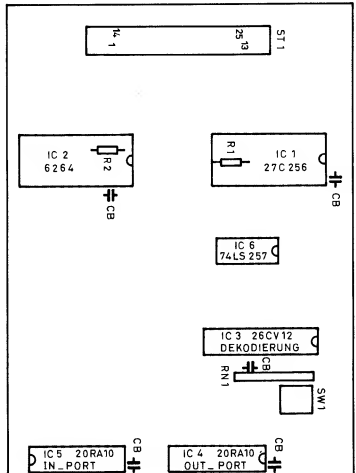

Bild 15. Die Speicherplatine der dCPU-4 bietet zusätzlich acht I/O-Ports.

```
CHIP      OUT_PORT  GAL20RA10

/PL       /WR_OUT   A0        NC    NC    NC    D0    D1    D2
D3        NC        GND

/OR       NC        NC        O0    O1    O2    O3    O4    O5
O6        O7        VCC

O0        :=        D0 * /A0 + O0 * A0      ; Untere Bits: A0 = 0
O0.CLCK   =         /WR_OUT

O4        :=        D0 * A0 + O4 * /A0      ; Obere Bits: A0 = 1
O4.CLCK   =         /WR_OUT
```

Listing 7. Ausgabeport auf der Speicherplatine (IC4, 20RA10).

```
CHIP    IM_PORT              GAL20RA10

/PL     NC          I0       I1          I2    I3    I4    I5    I6
I7      NC          GND

/OE     D3_1        D2_1     D1_1        D0_1  D3_0  D2_0  D1_0  D0_0
NC      A0          VCC

D0_0  =  I0 ; Die Ausgaenge treiben den Datenbus, und
D0_0.TRST = /A0  ; zwar doppelt, wobei die eigentliche

D0_1  =  I4  ; Auswahl durch A0, das Treiben der Daten-
D0_1.TRST =  A0  ; leitung durch /OE geschieht


CHIP      Dekodier-IC  GAL26CV12

A1          A2          A3          A4          A5          A6          VCC
A7          A8          A9          A10         A11         /RD         /WR
A214        A213        A212        A211        /CS_EPR     /CS_RAM     GND
/WR_OUT     /RD_IN      P3          P2          P1          P0          NC

A211  =  P0  ; nur einfaches Durchschalten; dies bewirkt, dass
A212  =  P1  ; jeder Codebereich 2 kByte gross sein kann, wobei
A213  =  P2  ; allerdings die Schaltung am EPROM eine A214  =  P3
             ; Beschraenkung auf prinzipiell 1 kByte ergibt.

CS_EPR  =  A8 * /A9 * /A10 * /A11 * RD  ; bierdurch ergeben sich
                                        ; 256 Bytes Code

RD_IN   =    A1 * A2 * A3 * A4 * A5 * A6 * A7    * A8 * /A9 * /A10
* /A11 * RD  ; in den obersten 2 Bytes

WR_OUT  =  A1 * A2 * A3 * A4 * A5 * A6 * A7    * A8 * /A9 * /A10
* /A11 * WR  ; in den obersten 2 Bytes

CS_RAM  =  A8 * /A9 * /A10 * /A11 * /RD_IN * /WR_OUT
; in den unteren 254 Bytes befindet sich RAM
```

Listing 8. Eingabeport auf der Speicherplatine (IC5, 20RA10).

```
CHIP        Dekodier-IC GAL26CV12

A1          A2          A3          A4          A5          A6          VCC
A7          A8          A9          A10         A11         /RD         /WR
A114        A113        A112        A111        /CS_EPR     /CS_RAM     GND
/WR_OUT     /RD_IN      P3          P2          P1          P0          NC

A111    =     P0            ; nur einfaches Durchschalten; dies bewirkt, dass
A112    =     P1            ; jeder Codebereich 2 kByte gross sein kann, wobei
A113    =     P2            ; allerdings die Schaltung am EPROM eine
A114    =     P3            ; Beschraenkung auf prinzipiell 1 kByte ergibt.
CS_EPR  =    /A8 * /A9 * /A10 * /A11 * RD     ; bierdurch ergeben sich
                                               ; 256 Bytes Code
RD_IN   =      A1 * A2 * A3 * A4 * A5 * A6 * A7
             * A8 * /A9 * /A10 * /A11 * RD    ; in den obersten 2 Bytes
WR_OUT  =      A1 * A2 * A3 * A4 * A5 * A6 * A7
             * A8 * /A9 * /A10 * /A11 * WR    ; in den obersten 2 Bytes
CS_RAN  =      A8 * /A9 * /A10 * /A11 * /RD_IN * /WR_OUT
             ; in den unteren 254 Bytes befindet sich RAN
```

Listing 9. Dekodierung des Speicherbereichs (IC3, 26CV12).

```
; Dieses Programm simuliert einen Unterprogrammsprung!
; Am Ausgabeport wird dabei eine 0 staendig durchgeschoben, ansonsten
; verlaeuft das Programm in einer Endlosschleife
#define PORT_LOW      $FB
#define PORT_HIGH     $FF
#define IW_1          $10
#define   PC_LOW      $80
START_3:
              LDAA    #$B           ; eine LBD soll leuchten!
              STAA    IW_1          ;
              LDAA    #$F           ;
              STAA    IW_1+1        ;
              SEC                   ; fuer ausgeschaltete LED
              LDAA    #*            ; Laden des Programmzaehlerstands
              ADDA    #$c           ; Offset von 12 fuer richtigen Ruecksprung!
              STAA    PC_LOW        ; Aber: Dies war leider nur ein 4-Bit Wert!
              LDAA    #$1           ; und dies die obere Haelfte, leider ohne
              STAA    PC_LOW+1 ; Assemblerunterstuetzung, muss also selbst
                                    ; berechnet werden
              JMP     UNTBR         ; Unterprogramm wird aufgerufen
              JMP     START_3       ; Endlosschleife
_UNTBR:
              LDAA    IW_1          ; Port setzen und naechsten Wert
              STAA    PORT_LOW ; vorbereiten
              ROLA                  ;
              STAA    IW_1          ;
              LDAA    IW_1+1        ; High Nibble
              STAA    PORT_HIGH     ;
              ROLA                  ;
              STAA    IW_1+1        ;
              JMP     [PC_LOW] ; Ruecksprung!
.BND
```

Listing 10. Assemblerroutine mit emulierten Unterprogrammsprung.

Assemblierung. Ausnahme der allgemeinen Regel: Das Fehlen einer Stack-Struktur läßt auch keine Unterprogrammsprünge zu. Dieses Fehlen kann jedoch durch einen Assemblertrick auf gute Weise ausgeglichen werden.

Eine Möglichkeit dazu zeigt Listing 10, in dem ein Assemblerprogrammabschnitt für die dCPU-4 dargestellt ist. Der Rücksprung aus der Routine – 'UNTER' genannt – wird in einer vereinbarten Stelle ($80 und $81) gespeichert, die für einen indirekten Sprungbefehl genutzt wird. Aus diesem Grund kann die Routine auch von mehreren Punkten erreicht werden, ohne daß ein festgelegter (direkter) Sprungbefehl dafür genutzt werden muß. Der Pferdefuß dieser Methode besteht darin, daß 8-Bit-Assembler normalerweise die 2 × 4-Bit-Adresse des PC nicht symbolisch laden können. Daher muß die Adreßberechnung zumindest für die High-Nibble-Adresse per Hand geschehen. Dies wäre natürlich mit einem echten 4-Bit-Assembler ausgeschlossen.

Damit ist der Phantasie der Programme, die neben dem geläufigen Assembler auch die Mikroprogramme erfaßt, (fast) keine Grenze mehr gesetzt. Im Prinzip können die Probleme nun dadurch gelöst werden, daß nicht mehr die Software, sondern zuerst die Mikroprogrammierung erstellt wird, auf der dann die 'klassische' Software basiert. Das Projekt dCPU-4 ist natürlich mehr ein Lehrstück, an dem prinzipiell alle Aktionen erkannt werden können.

Der nächste Schritt, die Synthese dieser Logik in einem CPLD oder FPGA, ist da schon wesentlich aufregender, was die erweiterten Möglichkeiten angeht. Die iCPU-4, die integrierte Version der dCPU-4, wird in einer der folgenden ELRAD-Ausgaben vorgestellt. Sie bietet einige wesentliche Erweiterungen gegenüber der diskret aufgebauten Version. Allerdings ist ihre Arbeitsweise von außen nicht mehr so leicht zu durchschauen. Weiterhin ist eine Beschreibung des Rechnermodells in der VHSIC Hardware Discription Language (VHDL) geplant. *pen*